[0072] In some embodiments, when the flash function table or flash rodata is accessed the returning data must be authenticated before it is used. Otherwise, an attacker could modify the function pointers or constants stored in the flash memory **306** to change the operation of the processor **308**. However, it is desirable to make the access latency of the function pointer table as small as possible so that firmware performance is not impacted. Accordingly, in some embodiments the instruction cache controller **326** performs the authentication operations and the instruction cache **328** may provide low-latency access on a cache hit.

[0073] The flash controller **304** may receive requests from the instruction cache controller **326**, the external interface module **322** and, as illustrated in **FIG. 2**, other processing components **222** (e.g., a NIC controller). In some embodiments these components arbitrate for access to the flash controller **304**.

[0074] If the instruction cache controller **326** has the grant of the flash controller **304** when the processor **308** (via the external interface module **322**) requests arbitration from the flash controller **304**, the TPM firmware may enter failure mode. Likewise, if the external interface module **322** has the grant when the instruction cache controller **326** requests the grant, the instruction cache controller **326** may assert the error signal to the instruction multiplexer **312**.

[0075] In some embodiments provisions may be made to avoid a deadlock scenario that may otherwise occur when the instruction cache controller **326** is prevented from requesting arbitration because the external interface module **322** has been given the flash grant. For example, the designer may ensure that any functions that access the external interface module **322** are only placed in the on-chip ROM **310**. Since these functions are not placed in the flash function table **340**, the likelihood of simultaneous instruction cache controller and external interface module accesses to the flash controller may be reduced.

[0076] To support the access to the flash memory **306**, the processor **308** may be configured to stall when there is an instruction cache miss during a flash function table (or rodata) read. The processor **308** stalls until the data is available. Also, on a flash instruction miss, the processor **308** may stall until the instruction is available.

[0077] To support the above accesses to the data bus **316**, the processor **308** may be configured to stall when there are multi-cycle requests on the data bus **316**. In some embodiments all data bus accesses take just one cycle with the exception of the flash function table read.

[0078] The memory map of TPM **302** may be configured to support the secure code components discussed herein. For example, in some embodiments unique address ranges are assigned for the OTP memory **332**, the cryptographic processor(s) **330**, the external interface module **322**, the instruction cache controller **326**, the instruction multiplexer **312**, the instruction ROM **310**, the flash memory **306** and secure assurance logic (not shown).

[0079] Referring now to the flowcharts of **FIGS. 4-8**, various examples of operations that may be performed by one embodiment of a system constructed in accordance with the invention will be discussed in more detail. **FIG. 4** relates to operations that may be performed when the system is manufactured. **FIG. 5** relates to operations that may be performed when the system boots up. **FIG. 6** relates to data and instruction access operations the system may perform. **FIGS. 7 and 8** relate to operations that may be performed when new and/or modified functions are loaded into an external memory.

[0080] In **FIG. 4**, as represented by block **402**, initially a code designer defines the original program code for the TPM. This code may include, in addition to typical TPM code, code that supports the secure code load mechanism and associated secure code components. For example, the boot code for the TPM may include provisions as discussed below that support the secure code load mechanism. In addition, the program code for the TPM may include function calls that are used to securely load new and/or modified code. The secure code scheme may be designed so that a designer is largely unaffected when coding TPM commands. For example, in some embodiments only boot and command switch firmware may be modified or added to support secure code load. Also at this stage, security routines that are to be stored in the OTP memory may be defined.

[0081] Once the code has been written, the code is processed by a compiler to generate the machine code that is to be loaded into data memory. Several secure code load specific operations are discussed in conjunction with blocks **404-410**. In some embodiments, several of these operations may be implemented through the use of scripts that are provided as extension of the compiler. Hence, the operations described below may be partially or completely automated. Alternatively, as discussed below, in some embodiments a designer may have control over some of these operations.

[0082] As represented by block **404** the designer determines which functions defined in the program code will be stored in internal memory (e.g., on-chip ROM) and will, consequently, be entered in the on-chip function table. Here, the designer may generate a file (e.g., romOnlyFunctions.txt) that specifies whether a function only resides in ROM.

[0083] In addition, the designer may generate a file (e.g., secureCodeCommands.txt) that contains a list of commands that are called during secure code load. The designer may recursively parse the call tree for these commands. All secure code load functions that are called and are not listed in romOnlyFunctions.txt are placed in the on-chip function table.

[0084] The designer may then determine the function partitioning between instruction ROM and flash memory. For example, the designer may generate a file (flashFunctions.txt) that specifies which functions reside in external memory (e.g., flash memory). After applying flashFunctions.txt, the designer may calculate how much code is to be placed in flash memory. If the code does not fit in ROM, the designer may find infrequently used functions and allocate them to flash memory. Functions specified in romOnlyFunctions.txt are not moved to flash. The designer may thus cause two assembly files to be generated: one for the internal ROM and one for flash memory.

[0085] The designer also may determine the order of the functions to be stored in on-chip ROM. In some embodiments functions are ordered to maximize locality.

[0086] The designer also may determine the order of the functions to be stored in flash memory, if applicable. Func-